Practical Assignment 1 - 2023/2024 PFL

Splut

***Group**: Splut_7*

***Members**:*

- *Guilherme Correia da Silva Coutinho, up202108872 - 50%*
- *Xavier Ribeiro Outeiro, up202108895 - 50%*

Installation and Execution

To correctly execute the game in both Linux and Windows environments, follow these steps:

- Step 1: Install SICStus Prolog 4.8.
- Step 2: Download the zip file `PFL_TP1_T05_Splut7.zip` and decompress it.
- Step 3: Move to the `src` directory and inside it consult the file `main.pl` using the command line (typing the command `consult('main.pl').`) or using Sicstus Prolog's UI.

```
?- consult('main.pl').
```

- Step 4: To start the game, run the predicate `play/0`.

```
?- play.
```

Description of the Game

Splut! is a 2 player abstract board game. It is played on a diamond-shaped board. Each player starts with three pieces: a Stonetroll, a Dwarf, and a Sorcerer. The objective is to eliminate the opposing Sorcerer by throwing a Rock at his head, resulting in the removal of the entire team.

Players take turns, with each typically making three steps per turn, except for the first and second players who have limited steps initially. TThe pieces have different kind of moves:

- Stonetrolls: If a rock is right behind them they can pull it in the direction of their move and and throw Rocks by moving in to it space, they chose the direction for the thrown and them the rock moves in straight lines until it its an obstacle.
- Dwarves: Dwarves can push consecutive rows of pieces in a straight line during their moves.
- Sorcerers: Sorcerers can levitate Rocks that make the same move as them.

Splut! requires careful planning and is a challenging board game that rewards clever and tactical gameplay. The game rules cand be found here (https://www.iggamecenter.com/en/rules/splut#board).

Game Logic

Internal Game State Representation

The Board

The game **Splut!** has a diamond shaped board, with 9 rows and 9 columns:

```
|   |   | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |   |   |

|   |   |   |   |   |   |   |||||   |   |   |   |   |   |

| 1 |   |   |   |   |   |///|_r_|\\\|   |   |   |   | 1 |

| 2 |   |   |   |   |///|tr2|dw2|sr2|\\\|   |   |   | 2 |

| 3 |   |   |   |///|   |   |   |   |   |\\\|   |   | 3 |

| 4 |   |   |///|   |   |   |   |   |   |   |\\\|   | 4 |

| 5 |||||_r_|   |   |   |   |   |   |   |   |_r_||||| 5 |

| 6 |   |   |\\\|   |   |   |   |   |   |   |///|   | 6 |

| 7 |   |   |   |\\\|   |   |   |   |   |   |///|   |   | 7 |

| 8 |   |   |   |   |\\\|sr1|dw1|tr1|///|   |   |   | 8 |

| 9 |   |   |   |   |   |\\\|_r_|///|   |   |   |   | 9 |

|   |   |   |   |   |   |   |||||   |   |   |   |   |   |

|   |   |   | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |   |   |
```

This board is created using the predicate `initialize_board/0` which asserts a total of 169 cells (13x13). A `cell/3` stores the Row and Column it belongs to and the Symbol it represents, which can be a whitespace or a piece of the game. Only 81 cells represent the actual board squares, the rest of them are used for user interface purposes like marking the number of the rows and columns and representing the limits of the board with slashes `/`, pipes `|` and blackslashes `\`.

Current Player

At the start of the game, the two players are set, using the predicate `initialize_players/0`. The predicate that starts a move of a player is `next_turn/6`. In this game, the player that starts has the right to only one move and the other player to only two moves, so we call this predicate one time for `player1` and two times for `player2`. Throughout the game, depending on the mode we are playing, each predicate recieves as an argument, which is the player of that turn. For example, the predicate `playing/1` that manages the player vs player mode, recieves the argument `PlayerTurn`. This predicate calls other predicates three times each loop it makes, which represents the three moves that each player have in this game. This predicate is called recursively.

Captured Pieces

In this game, the pieces are not captured, but rather taken away from the board. It can happen when a `Rock` falls on top of a `Dwarf`, which "kills" him, consequently being taken out of the bord. Apart from this situation, the other pieces are only taken away from the board when a player's `Sorcerer` is "killed", which happens when a `Rock` is thrown at him by a `Troll`. If this happens, the player loses and all their pieces are taken away from the board.

Initial State Of The Game

As mentioned earlier, the game starts when the predicate `play/0` is called. This predicate calls several functions that set the initial data of the game and asks the user which mode he wants to play in (player vs player, player vs computer or computer vs computer). Depending on the user's choice, the game will call a different predicate for each.

Intermidiate State Of The Game

Depending on the mode that was chosen, there's a predicate that is being called recursively while the game is being played. These predicates are `playing/1`, `playing_vs_computer/2` and `computer_vs_computer_playing/1`. During the game, a player can use the `Troll` to throw a `Rock`, but following the rules, when this ability is used, the player's turn comes to an end, starting the other player's turn immediately. For this, we call a specific predicate for each game mode after each move of a player, to check if they have thrown a rock. An example of these predicates is `check_turn_change/3`.

Game State Visualization

After each player move, the predicate `display_game/1` is called. This predicate verifies that the `PlayerTurn` argument recieved is a factual player, prints the board to the terminal as shown earlier calling the predicate `print_board/0` (which calls other auxiliar predicates printing a row at a time) and writes the current player.

When the chosen game mode requires a player, there are multiple menus that are displayed in the terminal an require the user to input a choice based on the choices on the menu displayed. All of the inputs are integers. Some examples of these menus are:

```
Choose a piece to move:
1. tr1
2. dw1
3. sr1
```

```
Choose a direction to move:
1. up
2. down
3. left
4. right
```

```
In what direction do you want to throw the rock?
1. up
2. down
3. left
4. right
```

```
Do you want to pull the rock?
1. Yes
2. No
```

Move Validation and Execution

After the user provided an input specifying which piece he wants to move, `move/8` is called. This predicate checks if facts are true like verifying that the user and the piece selected exists, if this piece is on the board, calls another predicate for this piece's specific moving algorithm, and after it finishes the algorithm and all the variables are set, it asserts the piece's new position on the board. Each piece has a different predicate for handling its possible options for moving. Inside each predicate, a list of possible moves is generated. With that, we ask the user which direction they want to move the piece to based on the possible ones. Some examples of these are `tr_move/6`, `dw_move/3` and `sr_move/6`.

List of Valid Moves

As mentioned before, each piece has a specific predicate that manages its possible moves and abilities. Inside those, there's a call for a auxiliar predicate that generates the list of valid moves that a piece can make. This algorithm checks the piece's adjacent squares and sees if they are empty. If that's the case, then that direction is added to the list of options that a user can choose to move the piece. Other decisions are made based on each piece's rule. For example, the `Troll` can go to a adjacent square that is not empty but is occupied by a `Rock`. If a user chooses that option, the `Troll` picks up the `Rock` and another menu show up asking for the user to select in what direction they want to throw the `Rock`. Keep in mind that the options that the rock is capable of being thrown in is generated by another auxiliar predicate called `check_rock_throw_options/2`. All of this is done in `check_tr_options/2`.

End of Game

When a `Sorcerer` is killed, the game is supposed to end. For that, we use the same predicate for checking if a player's turn has come to an end to also check if the game has ended. When the game does end, this predicate calls `game_over/1`, which prints the winner to the terminal and calls `halt/0` to terminate sicstus.

Game State Evaluation

After every move made, the predicate `check_turn_change/3` is called. This checks if by any reason or rule, the player turn has changed or the game has ended. This is done by checking the values of variables that are set throghout the algorithm of the game.

Computer Plays

Computer moves are randomly generated using the built-in `random/3` predicate and some predicates that we built like `random_select/2` which selects a random element from a list. All of these are generated in the same place that a user is asked what they want to do. This is done by managing an argument of the predicate `move/8` which is called `IsComputer`. If this argument is set to `1`, it means that the current player is a computer, so insted of prompting the user interface predicates, it generates a random number between the choices available for that piece.

Conclusions

This work has proven to be a great way to learn both a new language and a different way of thinking, which was new to us. We believe that we have reinforced our prior knowledge and also expanded it.

On the other hand, due to the fact that we also have numerous projects for other curricular units, we had insufficient time to fulfill all the requirements and to develop the game more in the way we intended to.

We wanted to have implemented the Level2 of computer play, we thought on trying to always move troll to the closest position of the sorcerer line, or to try to move the rock to the closest position of the sorcerer line.

Given our limited knowledge of this language, which has proven to be one of the challenging factors, it also contributed to the lack of time being the major difficulty in this project.

Bibliography

The game rules were consulted here (https://www.iggamecenter.com/en/rules/splut#board).